Special issue on International Workshop on Networked Sensing Systems

Trans. of the Society of Instrument and Control Engineers
Vol.E-S-1, No.1, 33/42 (2006)

Approximate Causal Observer[†]

Sandeep S. Kulkarni * and Mahesh Arumugam *

In this paper, we focus on the problem of approximate causal delivery. This problem identifies the tradeoff between causal delivery and timely delivery of messages. Causal delivery requires that delivery of a message, say m, be delayed until all messages on whom m is causally dependent are delivered. By contrast, timely delivery requires that messages be delivered as soon as possible. In the context where messages could be lost and the *value* of messages decreases as the delay increases, the requirements of causal delivery and timely delivery are conflicting.

We show how a simple logical timestamp program can be used to obtain a solution for approximate causal observer. This solution is intended for systems that provide simple guarantees about the clock drift and about maximum delay of messages that are not lost. While $O(n^2)$ *unbounded* integers are required to implement *perfect* causal delivery, our solution uses only $O(log\ n)$ *bounded* space. Our solution permits a process to tradeoff between causal delivery and timely delivery, i.e., it allows the process to choose the level of causality violations it can tolerate (0% or more) and the time for which it will have to buffer the received messages. We also show that the information maintained by our program, although small, is important to provide such a tradeoff; we show that the number of causality violations increase by an order of magnitude if this information is not maintained. Finally, we show how our solution can be used to observe computations in sensor networks while providing a continuum where one can choose the size of the timestamps based on the acceptable level of causality violations.

Key Words: Causal delivery, Timeliness, Logical timestamps, Sensor networks

1. Introduction

The ability to *observe* distributed computations is one of the important problems in many systems. Such a problem arises in several contexts. For example, consider an application in sensor networks (e.g., MICA motes [1]) where a group of sensors need to track a moving object (e.g., [2]). In such a system, the sensors communicate their observations about the object they are tracking with each other. However, due to limited memory/computing power and small size, a sensor cannot provide human readable output. Hence, these applications typically include a more powerful *visualization unit* (e.g., a PC) that is responsible for providing the required human readable output. Thus, the visualization unit *observes* the communication among sensors and uses its high computing power/memory to display/interpret the communication among sensors.

Other applications of such observer occur in debugging of distributed programs. In these applications, the debugger *observes* the communication among processes to study the behavior of the underlying program, and to identify potential violations of specifications. Yet another application includes monitoring distributed programs. Again, in such an application, the monitor receives the copies of (relevant) messages that are sent by the underlying processes. The monitor uses these messages to examine the behavior of the underlying program.

Since the order in which the observer receives messages may be different from the order in which the communication occurred in the underlying system, the observer needs to reorder messages consistently. While it may be impossible to recreate the exact scenario that occurred in the underlying system, it is desirable to obtain at least a consistent view at the observer. One way to obtain such consistent view is to ensure that the observer *delivers* the messages in a causal order.

One additional requirement for *online* observers is that they should be *timely*, i.e., the observer needs to reconstruct the underlying computation quickly. As an illustration, in the sensor network example given above, the visualization of the underlying computation should be in real-time and, hence, the visualization unit must act quickly on the messages received so that the visualization does not significantly lag behind the original computation. Likewise, if monitoring is used to identify potential errors in

[†] Presented at INSS2004 (2004.6)

* Computer Science and Engineering, Michigan State University, East Lansing MI 48824 USA

(Received November 12, 2004)
(Revised April 7, 2005)

the system computation then the monitor needs to identify the potential errors quickly.

It is easy to observe that there is a potential conflict between achieving causal delivery and delivering messages quickly. Causal delivery requires that a message be buffered until all messages that causally depend on it are delivered. However, timely delivery requires that a message be delivered as soon as possible. Since these goals are contradicting, it is necessary to develop protocols where the observer can choose the level of causality violations it can accept to ensure timely delivery of messages. In other words, if the observer introduces additional delays during delivery of messages then the number of causality violations should be reduced. We call this the problem of *approximate causal delivery*. And, an observer that provides approximate causal delivery is called *approximate causal observer*.

Clearly, based on the result in [3], one cannot design solutions for such approximate causal delivery in pure asynchronous systems where process speeds, process clocks and message delays are arbitrary. In other words, the underlying system must provide some simple guarantees that enable the observer to obtain the tradeoff between causal delivery and timely delivery. We focus on two simple guarantees from the underlying system: The first guarantee is related to the clocks; we require that the system provide a bound, say ϵ, such that the clock drift between different processes (underlying processes as well as the observer) in the system is bounded by ϵ. This guarantee can be met by using GPS clocks, network time protocol, atomic clocks or clock synchronization programs (e.g., [4–6]). The second guarantee relates to the message delays; we require that messages that reach their destination do so within some bound, say δ. This guarantee can be met by using protocols that characterize messages as being timely or late (e.g., [7]).

One can use solutions such as matrix clocks [8] to solve the problem of approximate causal delivery. However, this approach suffers from four problems; for one, matrix clocks do not use the underlying physical clock and, hence, cannot easily handle timely delivery. For two, the protocol in [8] cannot handle lost messages; all subsequent messages that causally depend on the lost message become undeliverable. Thirdly, the size of the timestamp used in [8] is $O(n^2)$ where n is the number of processes in the system. Such a large size could be especially problematic in systems where the number of processes is large. Finally, in [8], as the computation proceeds, the size of the timestamps grows without a bound. While solutions in [9, 10] deal with the first two problems, these solutions still suffer from the overhead of timestamps whose size is quadratic in the number of processes and whose size grows unbounded as the computation proceeds.

Another related work on causal delivery in sensor networks is [11] where a temporal message ordering service is proposed. In this approach, the senor network is modified to achieve temporal message ordering at base station. Towards this end, they send each message by multiple routes, one or more short and one or more long. By contrast, in our work, we do not assume such multiple messages. In fact, we do not modify the underlying communication in the sensor network. The reordering is done only at the base station. It follows that our approach allows one to make use of any optimizations that can be performed in the routing layer.

With this motivation, in this paper, we focus on adapting the algorithm in [12] to obtain approximate causal delivery. This algorithm has several helpful features; for one, the size of the timestamps is bounded, i.e., it does not grow as the computation proceeds. Also, the size of the timestamps is proportional to the system guarantees, i.e., if the system guarantees are improved then the size of the timestamps is reduced. Moreover, the size of the timestamps is $O(\epsilon \ log \ n \ + \ log \ \delta)$ where n is the number of processes in the system.

Contributions of the paper. In this paper, we present two algorithms for approximate delivery. The first algorithm is based on delivering the messages earlier than that prescribed by the algorithm in [12]. Based on the ability to tolerate causality violations, this algorithm allows the observer to reduce the delivery time of messages. We also show that a simple modification to the first algorithm, checking messages that the observer has received to determine if they might violate the requirements of causal delivery, allows us to reduce the number of causality violations further. Also, we study the effect of changing various parameters (clock drift, message delay, and message rate) on approximate causal delivery.

We also show that even though the timestamps used in these algorithms do not have all the information necessary for detecting causality violations, the information maintained by the algorithms is highly valuable in reducing the number of causality violations. To illustrate this, we show that if only physical clock is used to obtain causal delivery then the number of causality violations increase significantly. Thus, small additional information maintained by the algorithm plays an important role in reducing the number of causality violations.

Finally, we show that the timestamp provides a continuum, i.e., the application developer can choose the size of timestamp by considering the number of causality violations and message delay the application can handle. In other words, if the overhead of the timestamps is too large, the developer can reduce the size of the timestamp at the cost of increased causality violations and/or increased delay.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we present our system model. Then, in Section 3, we present the algorithm for causal delivery in [12] and recall its relevant properties. In Section 4, we present our approaches for approximate causal delivery. In Section 5, we state our simulation model. And, in Section 6, we show the effect of various parameters on approximate causal delivery of messages and also show that the application can choose the size of the timestamp depending on the level of causality violations it can tolerate. Finally, in Section 7, we make concluding remarks and discuss future work.

2. System Model

A distributed system consists of finite set of processes which communicate via message passing. For simplicity, we do not distinguish between the processes and the processors they run on. Each process j has a physical clock $rt.j$.

As discussed in Introduction, in the absence of faults, a distributed system must provide some guarantees that will enable the observer to obtain a tradeoff between causal delivery and timely delivery. In the presence of faults, these guarantees may be violated temporarily. We focus on the following two guarantees about the bound on maximum clock drift (for example, using [6]) among different processes (sensors) and the bound on message delay.

Guarantees of the distributed system.

G1. The value of $rt.j$ is non-decreasing, and at any time, the difference between the clock values of any two processes is bounded by ϵ. In other words,

$$\forall j,k : |rt.j - rt.k| \leq \epsilon$$

G2. Let m_j be a message sent by process j to k. Also, let st_m denote the clock value of j when j sent m_j, and let rd_m denote the clock value of j when k received m_j. We require that k should receive m_j within time δ unless m_j is lost. In other words,

$$((rd_m \leq (st_m+\delta)) \quad \vee \quad rd_m = \infty)$$

Notation. A distributed system instantiated with parameters ϵ and δ is denoted as $ds(\epsilon, \delta)$.

Execution of a process consists of a sequence of events; an event can be a local event, a send event, or a receive event. In a local event, a process neither receives nor sends a message. In a send event, a process sends one or more messages, and in a receive event, a process receives one or more messages. For simplicity, we assume that one clock tick of j corresponds to at most one event at process j. Note that, we can weaken this assumption so that one clock tick corresponds to at most K events, where K is any constant.

We assume that there is a special *observer* process in the system. A copy of relevant messages sent by any process is also sent to the observer. The observer buffers the messages and delivers them in such a way that the number of causality violations is acceptable. Note that we do not assume that the observer can precisely determine causal relations between two messages.

Notation. In this paper, we use i, j, k and l to denote processes. We use e, f and g to denote events. Where needed, events are subscripted with the process at which they occur, thus, e_j is an event at j. We use m to denote messages. Messages are subscripted with the process that sends the message. Thus, m_j is a message sent by j.

3. Logical Timestamps and Causal Delivery

In this section, we present the algorithm for logical timestamp (Section 3. 1) and causal delivery (Section 3. 2) from [12]. We use the causal delivery algorithm in Section 3. 2 for achieving approximate causal delivery.

3. 1 Logical Timestamp

Before presenting the program, we define the notion of *happened-before*, $\longrightarrow$ among events.

Happened-before. The happened-before relation [13] is the smallest transitive relation that satisfies, for any events e, f, $e \longrightarrow f$ if (1) e and f are events on the same process and e occurred before f, or (2) e is a send event in one process and f is the corresponding receive event in another process. □

Solution to logical timestamp. In the solution to the logical timestamps proposed in [12], the timestamp of an event e_j at process j is of the form $\langle rt.e_j, c.e_j, kn.e_j\rangle$, where $rt.e_j$ denotes the physical clock value of j when e_j was created. The variable $c.e_j$ denotes the difference between the knowledge j had about the maximum clock value in the system and the physical clock value of j. The variable $kn.e_j$ is an array of size 2ϵ. The variable $kn.e_j[t], -\epsilon \leq t < \epsilon$, captures the knowledge about the number of events f such that $r.f = r.e_j + t$ and $f \longrightarrow e$.

Each process j in the system maintains $rt.j$, $r.j$, $c.j$ and $kn.j$. (The algorithm works correctly even if j maintains $(rt.j \ mod \ B)$ instead of $rt.j$, where $B \geq \epsilon+\delta+1$. Thus, the space required for maintaining $rt.j$ is bounded.) The variable $rt.j$ represents the physical clock value at j and $\langle r.j, c.j, kn.j\rangle$ represents the timestamp of the last event at j. The initial and update rules for different events are presented in Figure 1. Note that, for simplicity of presentation, we assume $kn.j[t] = 0$ if $t < -\epsilon$ or $t \geq \epsilon$.

```
Initially:
    $rt.j, r.j, c.j = 0,$
    $\forall t : t \neq 0 : kn.j[t] = 0, kn.j[0] = 1$

Local event $e_j$ or
Send event $e_j$ (message being sent is $m_j$)
    $c.j := max(0, r.j + c.j - rt.j)$
    $\forall t : -\epsilon \leq t < \epsilon : kn.j[t] := kn.j[t + rt.j - r.j]$
    $kn.j[0] := kn.j[0] + 1$
    $r.j := rt.j$
    $r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$
    if $e_j$ is a send event then
        $r.m_j, c.m_j, kn.m_j := r.j, c.j, kn.j$

Receive event $e_j$ (message $m$ received
with timestamp $\langle r.m, c.m, kn.m\rangle$)
    $c.j := max(0, r.j + c.j - rt.j, r.m + c.m - rt.j)$
    $\forall\ t : -\epsilon \leq t < \epsilon : kn.j[t] :=$
      $max(0, kn.j[t + rt.j - r.j], kn.m[t + rt.j - r.m])$
    $kn.j[0] := kn.j[0] + 1$
    $r.j := rt.j$
    $r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$
```

Fig. 1 Logical timestamp program

Comparing timestamps. Let $\langle r.e_j, c.e_j, kn.e_j\rangle$ and $\langle r.f_k, c.f_k, kn.f_k\rangle$ be two timestamps. The *less* function for comparing timestamps based on the logical timestamp program in Figure 1 is as follows:

$$less(\langle r.e,_j\ c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$$

iff

$$(r.e_j + c.e_j, kn.e_j[c.e_j], kn.e_j[c.e_j - 1], \ldots, kn.e_j[c.e_j - \epsilon + 1], j)$$
$$< \text{// lexicographic comparison}$$
$$(r.f_k + c.f_k, kn.f_k[c.f_k], kn.f_k[c.f_k - 1], \ldots, kn.f_k[c.f_k - \epsilon + 1], k)$$

In the above comparison, kn values are compared only when $r.e + c.e$ equals $r.f + c.f$. Thus, $kn.f[c.f]$ is compared with $kn.e[r.f + c.f - r.e](= kn.e[c.e])$. Since $kn.f[t]$ denotes the knowledge about events at $r.f + t$, the comparison of kn values allows us to determine if f_k was aware of more events than e_j.

Properties of the logical timestamp program. The logical timestamp program presented above has the following properties. (We refer the reader to [12] for proof.)

- $\forall e, f :: e \longrightarrow f$
  $\Rightarrow less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$.
- The value of $c.e$ is less than ϵ and the value of each element in $kn.e$ is less than n, where n is the number of processes in the system. Hence, the space needed by the timestamp is $O(\epsilon\ log\ n\ +\ log\ \delta)$. Further, it does not grow as the computation proceeds.

3. 2 Causal Delivery Program

The causal delivery program proposed in [12] is as follows: Whenever a process j receives a message m, j buffers the message until $delcond(m, j) = (rt.j = r.m + c.m + \delta + \epsilon)$ is satisfied. As soon as the $delcond(m, j)$ is satisfied, the message is delivered. If two or more messages satisfy the delivery condition simultaneously then process j determines the causal relation among the messages and delivers them accordingly. If m_j and m_k satisfy the delivery condition simultaneously and $less(\langle r.m_j, c.m_j, kn.m_j\rangle, \langle r.m_k, c.m_k, kn.m_k\rangle)$ is *true*, then m_j is delivered before m_k.

Properties of the causal delivery program. The causal delivery program presented above has the following properties (cf. [12]).

- If process j sends a message m when its physical clock value was $r.m$ then the message would be delivered before the physical clock value of j reaches $r.m + \delta + 3\epsilon$.
- If two messages m_1 and m_2 such that $send(m_1) \longrightarrow send(m_2)$ arrive at any process j then m_1 is delivered before m_2.
- The causal delivery program is stabilizing fault-tolerant [14], i.e., starting from an arbitrary state —that may be reached by temporary violation of system guarantees or improper initialization of processes or transient state corruption of processes or message loss—, it eventually recovers to states from where its specification is satisfied.

4. Approaches for Approximate Causal Delivery

The algorithm presented in Section 3. 2 uses the delivery condition $delcond(m, j)$ to deliver a message m to process j. This condition is necessary for correctness, i.e., to ensure *all* messages are delivered in causal order. In other words, there exists messages for which this condition is optimal.

In the algorithm in Section 3. 2, message m is delivered at process j when $rt.j = r.m + c.m + \delta + \epsilon$. Thus, $c.m + \delta + \epsilon$ is the approximate delay in obtaining causal delivery. We consider the case where messages are delivered before this delivery condition is satisfied. However, instead of choosing fixed values for the reduced delay, we let the delay be proportional to the underlying system guarantees and any other information that m carries.

In the algorithm in Section 3.2, $r.m + c.m$ captures the knowledge that the sender of m had about the maximum clock in the system. Also, ϵ and δ depend on the underlying system guarantees. Hence, we let the reduced delay to be a certain percentage of the delay incurred while ensuring causal delivery. Thus, the actual delay incurred by messages depends on the underlying system guarantees (ϵ, δ) and the knowledge $(c.m)$ that m had 'about the future'.

Based on this approach for reducing the delay, we present two algorithms for approximate causal delivery: (1) deliver after partial wait and (2) check before delivery.

Deliver After Partial Wait (DAPW). In this algorithm, we use the following delivery condition: $delcond(m, j) = (rt.j = r.m + ¢(c.m + \delta + \epsilon))$, where $0\% < ¢ < 100\%$. Thus, $¢ = 0\%$ means that the messages are delivered to the observer when the clock of observer is at least $r.m$, or as soon as the message arrives at the observer, whichever is later. And, $¢ = 100\%$ means that the messages are delivered in perfect causal order. Thus, by using different values for ¢, the application can choose the delay in delivery.

Check Before Delivery (CBD). In DAPW, whenever a message, say m_1 is about to be delivered to process j, if there is a casually related message m_2 such that $send(m_2) \longrightarrow send(m_1)$ is true and m_2 is scheduled for delivery at a later time than m_1 then a causality violation is inevitable. Hence, we propose our second algorithm that checks the queue to determine causally related messages. Specifically, whenever message m_1 is about to delivered at process j, j checks the message queue to determine if there is any message m_2 such that $less(\langle r.m_2, c.m_2, kn.m_2\rangle, \langle r.m_1, c.m_1, kn.m_1\rangle)$ is *true*. If there exists such a message m_2 then j sets the delivery time of m_1 as $delcond(m_1, j) = delcond(m_2, j)$. If there are no such message then m_1 is delivered based on the DAPW algorithm.

5. Simulation Model

Our simulation model consists of n ordinary processes and one special process (observer). The ordinary processes communicate with each other. Every message sent by an ordinary process is also sent to the observer. Now, we show how our simulation model ensures system properties stated in Section 2.

Ensuring $G1$. At each step of the simulation, one process is selected at random based on a uniform distribution of $n + 1$ processes (i.e., n ordinary process and a special observer process). The selected process (say, j) can increment its physical clock $(rt.j)$ and send messages to other processes. The simulation program ensures $G1$ by selecting another process from the uniform distribution if incrementing $rt.j$ leads to violation of $G1$.

Ensuring $G2$. Whenever a process sends a message, the destination receives the message within $x, 0 \leq x \leq \delta$, unit(s) of time, thereby ensuring $G2$. Message delay is determined using a normal distribution $N(\mu, \sigma)$, where μ is the mean delay and σ is the standard deviation of the delay. In our simulations, we use $N(\frac{\delta}{2}, \frac{\delta}{4})$ (approximately 95% messages are received in $[0 \ldots \delta]$) and $N(\frac{\delta}{4}, \frac{\delta}{8})$ (approximately 95% messages are received in $[0 \ldots \frac{\delta}{2}]$) for message delay. If the random delay from the distribution is greater than δ, we treat it as a lost message. Since the message delay cannot be less than 0 in a real system, if the random delay from the distribution is less than 0, we choose another random delay from the same distribution.

After the system properties are met, the selected process can increment its physical clock and send messages to other processes.

Implementing ***message rate*****.** Whenever a process (say, j) increments its physical clock, it sends a message to other processes with certain probability. We implement this using *message rate*. Process j chooses a random number between 1 and 1/*message rate*. If the random number is 1, j can sends a message to another process. Also, whenever a process sends a message to another process, it sends a copy of the message to the observer.

6. Simulation Results

For our simulation, we developed an event simulation program in Java. The program takes number of ordinary processes, ϵ, δ, message rate, the mean of message delay, the standard deviation of message delay and the type of algorithm as input. We conducted experiments for $\delta = 10$ with the following values of ϵ: 5, 10, 20, and 30. Similarly, we conducted experiments for $\epsilon = 10$ with the following values of δ: 5, 10, 20, and 30. Note that, we have not associated a unit for ϵ and δ. If $\epsilon = 5$ and $\delta = 10$, it can be used to represent a system where the maximum clock drift is $5ms$ ($10ms$) and message delay is $10ms$ ($20ms$), etc. Further, we find that the ratio $\frac{\epsilon}{\delta}$ is important than the individual parameters.

For these values of ϵ and δ, we use the following values for *message rate*: 0.5, 0.1, and 0.01. Likewise, we use the following values for ¢: 100%, 80%, 60%, 40%, 20%, and 0%. For each input, we perform at least 3 experiments to compute the causality violations. The results presented here are average of these experiments. For a given value of the input parameters, the percentage of causality violations in different experiments are similar.

In these experiments, we compute the number of causality violations at the observer as follows: For each m_1, we compute the number of messages delivered before m_1 (say, m_2) such that $send(m_1) \longrightarrow send(m_2)$ is true. We say that these messages violate backward causality. Likewise, for each m_1, we computer the number of messages delivered after m_1 (say, m_2) such that $send(m_2) \longrightarrow send(m_1)$ is true. We say that these messages violate forward causality. The number of causality violations is obtained by taking the average of messages that violate backward/forward causality.

To compute these causality violations, for each event/message, we also maintain vector timestamps [15, 16] in addition to the logical timestamps from Section 3. These vector timestamps identify the actual causal relation among events in the system. They are not used in any way to determine when messages are delivered.

The rest of the section is organized as follows. In Section 6. 1, we study of effect of changing ϵ on causal delivery of messages. Then, in Section 6. 2, we study the effect of varying δ on causal delivery of messages. Subsequently, in Section 6. 3, we present the effect of *message rate* on causal delivery. In Section 6. 4, we present the effect of *number of processes* in the system on causal delivery. Finally, in Section 6. 5, we present the effect of using partial timestamps and show that the application can choose the size of the timestamp depending on the level of causality violations it can tolerate.

6. 1 Effect of Maximum Clock Drift

The effect of ϵ on causal delivery of messages using DAPW and CBD is shown in Figure 2. The graphs show the number of causality violations as a function of the percentage of delay, ¢, used in *delcond*. In these experiments, we use $\delta = 10$ and $message\ rate = 0.1$. The simulation consists of 10 ordinary processes and the special observer process.

DAPW. When the ratio $\frac{\epsilon}{\delta}$ is larger, the number of causally dependent messages for a given message m is large. Thus, for larger values of $\frac{\epsilon}{\delta}$, there is a higher probability that one or more of these messages are delivered before m. Hence, as $\frac{\epsilon}{\delta}$ increases, the number of causality violations increase (cf. Figures 2 (a) and 2 (c)).

When message delay is determined from the distribution $N(\frac{\delta}{4}, \frac{\delta}{8})$, 95% of the messages are received within $\frac{\delta}{2}$. By contrast, when message delay is determined from the distribution $N(\frac{\delta}{2}, \frac{\delta}{4})$, 95% of the messages are received in δ. Thus for $N(\frac{\delta}{4}, \frac{\delta}{8})$, the number of messages that causally depend on a given message is more than that for $N(\frac{\delta}{2}, \frac{\delta}{4})$. Hence, the probability of causality violations is more when the distribution $N(\frac{\delta}{2}, \frac{\delta}{4})$ is used for message delay (cf. Figures 2 (a) and 2 (c)).

For small values of $\frac{\epsilon}{\delta}$, there exists a threshold T such that, the causality violations increase suddenly when $¢ < T$. For example, in Figure 2 (a), for $\epsilon = 5$, there is a sudden rise in causality violations for $¢ < 40\%$. The number of causally related messages is less when the ratio $\frac{\epsilon}{\delta}$ is small. When $T \leq ¢ < 100\%$, the delay in delivery captures most of the causal relation among messages. When this delay is reduced (i.e., $¢ > T$), the messages are delivered faster and, hence, the causal relation among messages is not captured. For larger values of $\frac{\epsilon}{\delta}$, more causally related messages are present for a message m. Hence, the observer captures most causally related messages even when the delay in delivery is less.

CBD. From Figures 2 (b) and 2 (d), we observe that as $\frac{\epsilon}{\delta}$ ratio increases, the number of causality violations decrease. This result is exactly opposite to DAPW. In CBD, before delivering a message m_1, the message queue is checked to determine if there is any message, say m_2 such that m_1 causally depends on m_2. If there is such a message, CBD postpones the delivery of m_1. Thus, as $\frac{\epsilon}{\delta}$ ratio increases, CBD can detect/prevent most of the causality violations since there is higher probability that the message queue contains one or more causally related messages.

Contrary to the observation for DAPW, we note that the number of causality violations is less when CBD is used with message delay of $N(\frac{\delta}{4}, \frac{\delta}{8})$. This is due to the fact that there are more causally related messages for a given message m, and there is a high probability that at least one of them will be present in the message queue of the observer when m is about to be delivered.

Comparison. From Figure 2, we conclude that the number of causality violations in CBD are an order of magnitude less than that in DAPW. For small values of $\frac{\epsilon}{\delta}$, CBD performs almost similar to DAPW. When $\frac{\epsilon}{\delta}$ is small, the number of causally related messages for any message is less. Therefore, CBD has limited or no information in the message queue to detect/prevent causality violations, as opposed to the case where the ratio $\frac{\epsilon}{\delta}$ is large. Thus, for small values of $\frac{\epsilon}{\delta}$, it may be better to use DAPW and save the overhead of checking the queue

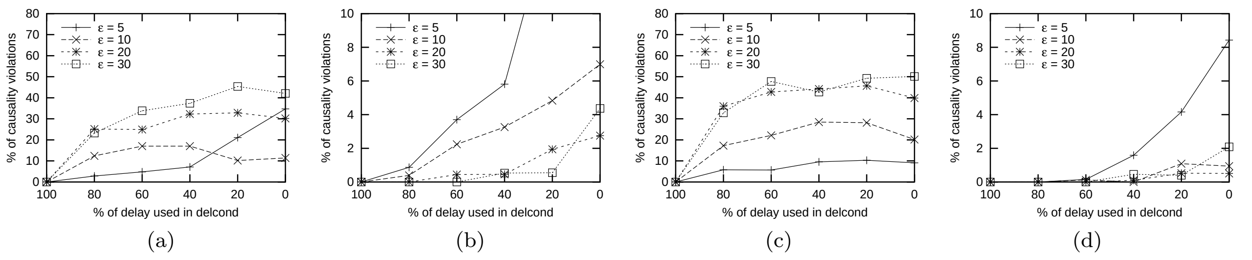

Fig. 2 Effect of ϵ on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

as one in CBD. Also, we note that the processing overhead with DAPW is significantly lower than that of CBD. Hence, for small values of $\frac{\epsilon}{\delta}$, we recommend DAPW. As the ratio $\frac{\epsilon}{\delta}$ increases, we prefer CBD, since the small addition in processing overhead reduces the number of causality violations considerably. (We observe similar results in Sections 6. 2 and 6. 3.)

6. 2 Effect of Maximum Message Delay

The effect of δ on causal delivery of messages using DAPW and CBD is shown in Figure 3. The graphs show the number of causality violations as a function of percentage of delay, $\cent$, used in *delcond*. In these experiments, we use $\epsilon = 10$ and *message rate* $= 0.1$. The simulation consists of 10 ordinary processes and the special observer process.

DAPW. From Figures 3 (a) and 3 (c), we observe that as $\frac{\epsilon}{\delta}$ increases, the number of causality violations increase. These results are similar to that in Section 6. 1.

Further, in Figure 3 (c), we observe that the number of causality violations is more when DAPW is used with message delay of $N(\frac{\delta}{4}, \frac{\delta}{8})$. Once again, these results are similar to that in Section 6. 1.

As mentioned in Section 6. 1, for small values of $\frac{\epsilon}{\delta}$, there exists a threshold T such that causality violations increase suddenly when $\cent < T$. For example, in Figure 3 (a), for $\delta = 20$ (respectively, $\delta = 30$), there is a sudden rise in causality violations when $\cent < 60\%$ (respectively, $\cent < 40\%$).

CBD. From Figures 3 (b) and 3 (d), we observe that as $\frac{\epsilon}{\delta}$ increases, the number of causality violations decrease. Once again, this is exactly opposite to DAPW.

6. 3 Effect of Message Rate

The effect of *message rate* on causal delivery of messages using DAPW and CBD is shown in Figure 4. The graphs show the number of causality violations as a function of percentage of delay used in *delcond*. In these experiments, we use $\epsilon = \delta = 10$. The simulation consists of 10 ordinary processes and the special observer process.

As the *message rate* increases, more causally dependent messages for a message m are present in the system. Thus, the probability of causality violations is higher. Further, the number of causality violations in CBD is significantly less than that in DAPW.

When δ is an overestimate of message delay (i.e., message delay of $N(\frac{\delta}{4}, \frac{\delta}{8})$), the number of causality violations in CBD is in the order of $0\% - 2\%$ (cf. Figure 4 (d)). This is due to the fact most messages arrive within $\frac{\delta}{2}$, and, hence, CBD has more information present in the message queue to detect/prevent causality violations before delivering a message.

Further, for small values of *message rate*, causality violations in DAPW and CBD are nearly equal. This is due to the fact that at low message rates, CBD has very limited information to exploit the messages in the queue. Further, as CBD incurs an additional overhead of processing the message queue, we expect that DAPW will be preferred for small values of *message rate*.

6. 4 Effect of Number of Processes

The effect of number of processes on causal delivery of messages using DAPW and CBD is shown in Figure 5. The results are for $\epsilon = \delta = 10$ and *message rate* $= 0.1$. We use the following values for the number of ordinary processes: 5, 10 and 50.

As the number of processes increases, more causally dependent messages for a message m are present in the system. Thus, the probability of causality violations is higher.

Furthermore, when δ is an overestimate of message delay (i.e., message delay of $N(\frac{\delta}{4}, \frac{\delta}{8})$), the number of causality violations in CBD is in the range of $0\% - 3\%$ (cf. Figure 5 (d)). Since most messages arrive within $\frac{\delta}{2}$, CBD detects/prevents most of the causality violations.

6. 5 Physical Clocks Vs. Partial Timestamps

In this section, we argue that the information maintained in CBD, although small, is important in reducing the number of causality violations. Towards this end, we compute the causality violations for the case where only

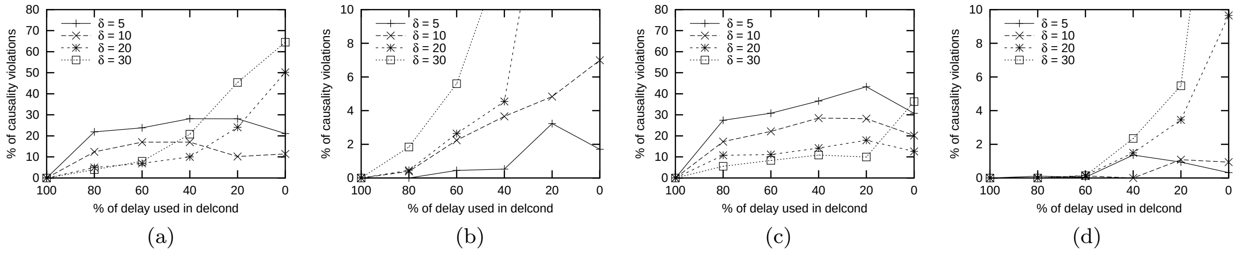

Fig. 3 Effect of δ on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2},\frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2},\frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4},\frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4},\frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

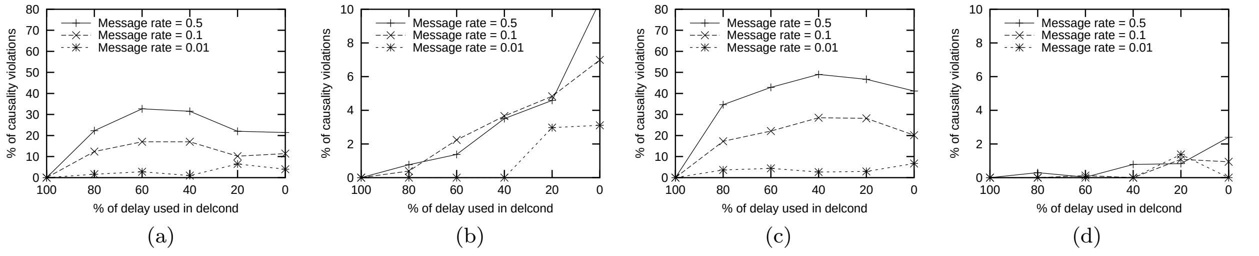

Fig. 4 Effect of *message rate* on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2},\frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2},\frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4},\frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4},\frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

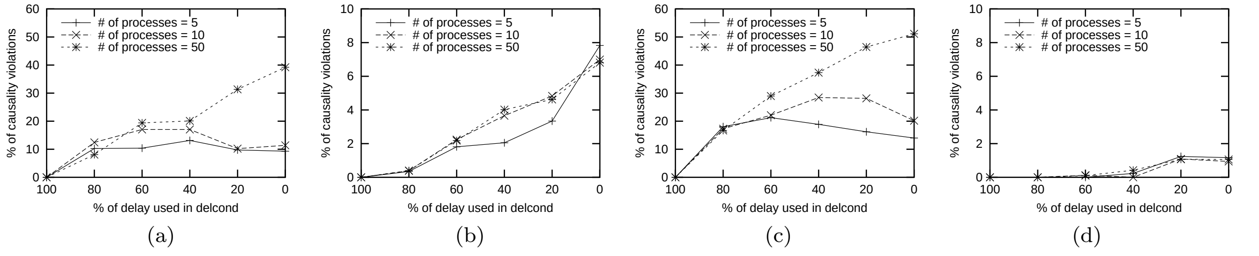

Fig. 5 Effect of *number of ordinary processes* on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2},\frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2},\frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4},\frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4},\frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

physical clock is used to determine when a message should be delivered. To obtain an implementation that uses physical clock alone, we set the c value and all elements in kn to 0. We call this algorithm DPC1. We also consider the algorithm DPC2 where the c value is used but kn values are reset to 0. Other points on this continuum can be obtained by maintaining a subset of the kn values in the timestamp.

Notation. In this section, by *"2 kn.e elements"* we mean that the simulation uses $kn.e_j[c.e_j]$ and $kn.e_j[c.e_j{-}1]$ elements instead of the $kn.e_j$ array for an event e_j. Similarly, by *"k kn.e elements"* we mean that the simulation uses the first k $kn.e$ elements.

Figure 6 shows the simulation results for $\epsilon = \delta = 10$, message rate$\,=0.1$ and 10 processes. (Figure 7 shows the results for 50 processes.)

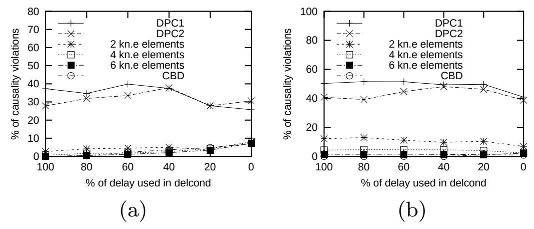

Fig. 6 Effect of using partial timestamps with 10 processes on (a) CBD with delay $N(\frac{\delta}{2},\frac{\delta}{4})$, (b) CBD with delay $N(\frac{\delta}{4},\frac{\delta}{8})$.

From Figure 6, we observe that using physical clocks alone for causal delivery of messages is not enough. Specifically, even when ¢$=100\%$, DPC1 and DPC2 have around 30%–50% of causality violations. And, maintaining just 2 $kn.e$ elements provides a significant reduction in number of causality violations (10−15%). Moreover, if we increase the number of $kn.e$ elements in the timestamp, the causal-

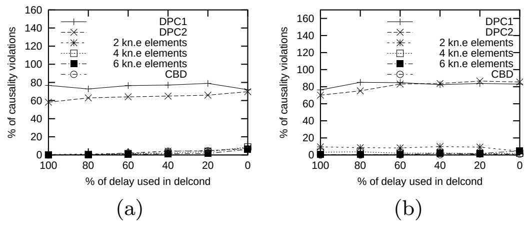

Fig. 7 Effect of using partial timestamps with 50 processes on (a) CBD with delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with delay $N(\frac{\delta}{4}, \frac{\delta}{8})$.

ity violations can be further reduced. Maintaining just 6 *kn.e* elements gives the same result as CBD. Thus, the timestamp provides a continuum in which the application developer can choose the size of the timestamps based on the requirements. This result is especially important in sensor networks. Specifically, in MICA motes [1], the payload size is just 29 bytes. Hence, the overhead in achieving approximate causal delivery should be small. Depending on the percentage of causality violations processes can handle and the overhead involved, the developer can choose an appropriate size for the timestamp. For example, choosing 2 *kn.e* elements (i.e., 4 bytes including *rt.j* and *c.j*) will result in $10 - 15\%$ causality violations (as opposed to $30 - 50\%$ causality violations when using the physical clocks alone). Thus, small additional information maintained in the timestamp plays a significant role in reducing the number of causality violations.

7. Conclusion and Future Work

In this paper, we presented a solution for approximate causal delivery. We discussed the effect of the parameters such as maximum clock drift, maximum message delay, and message rate on causal delivery of messages. We showed that by using physical clocks alone, the number of causality violations increase significantly. By adding new variables to the timestamp, the number of causality violations can be reduced. In other words, we showed that our solution provides a continuum such that the application developer can choose the size of timestamps used in the system based on the number of causality violations the application can tolerate. This result is especially useful in sensor networks, since the sensors are resource constrained and the size of the payload in a message is very limited (e.g., 29 bytes in MICA). From Section 6. 5, we note that maintaining just 2 *kn.e* elements (i.e., 4 bytes) provides a significant reduction in causality violations ($10 - 15\%$) compared to using physical clocks alone ($30 - 50\%$). Hence, causal delivery of messages at the base station can be achieved easily in sensor networks with a small message overhead. To our knowledge, this result is the first of its kind for providing approximate causal delivery in sensor networks. Moreover, DAPW and CBD preserve the self-stabilization [14] property of the algorithm in [12], i.e., starting from arbitrary initial states, the system recovers to states from where causal delivery is achieved. Hence, if the sensors are corrupted, our algorithm ensures that eventually approximate causal delivery is restored.

There are several possible extensions to this work. We are currently investigating the performance of our approach with trace data from experiments such as [17]. This allows us to study the effect of causality violations on large scale sensor network applications. Further, it allows the application developer to choose the size of the timestamps and the delivery time of a message. Another interesting extension to this work is to study of the effect of buffering time at the intermediate sensors.

Acknowledgments. This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF Equipment Grant EIA-0130724, and a grant from Michigan State University.

References

1) J. Hill and D. E. Culler. Mica: A wirleess platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.
2) M. Demirbas, A. Arora, and M. G. Gouda. A pursuer-evader game for sensor networks. *In Proceedings of the Sixth Symposium on Self-Stabilizing Systems (SSS), Springer*, LNCS: 2704:1–16, June 2003.
3) M. J. Fischer, N. A. Lynch, and M. S. Peterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.
4) J. M. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison. *In Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 486–493, 1992.
5) A. Arora, S. Dolev, and M. G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.
6) T. Herman. NestArch: Prototype time synchronization service. NEST Challenge Architecture, January 2003.
7) F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.
8) M. Singhal and N. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.* McGraw-Hill Publishing Company, New York, 1994.
9) R. Baldoni, M. Mostefaoui, and M. Raynal. Causal deliveries in unreliable networks with real-time delivery constraints. *Journal of Real-Time Systems*, 10(3):1–18, 1996.
10) F. Adelstein and M. Singhal. Real-time causal message ordering in multimedia systems. *International Conference on Distributed Computing Systems*, pages 36–43, 1995.
11) K. Römer. Temporal message ordering in wireless sen-

sor networks. *In Proceedings of the Second Mediterranean Workshop on Ad-Hoc Networks (MED-HOC NET)*, June 2003.
12) S. S. Kulkarni and Ravikant. Stabilizing causal deterministic merge. *In Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer*, LNCS:2194:183–199, October 2001.
13) L. Lamport. Time, clocks, and the ordering of events in a disributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
14) E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
15) J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.
16) F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.
17) A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y-R. Choi, T. Herman, S. S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier)*, 46(5):605–634, December 2004.

Sandeep S.kulkarni

Sandeep Kulkarni received his B.Tech. in Computer Science and Engineering from Indian Institute of Technology, Mumbai, India in 1993. He received his MS and Ph.D. degrees in Computer and Information Science from Ohio State University, Columbus, Ohio, USA in 1994 and 1999 respectively. He has been working as an assistant professor in Michigan State University, East Lansing, USA since August 1999. He is a member of the Software Engineering and Network Systems (SENS) Laboratory. He is a recipient of the NSF CAREER award. His research interests include fault-tolerance, distributed systems, group communication, security, self-stabilization, compositional design and automated synthesis. Contact him at sandeep@cse.msu.edu.

Mahesh Arumugam

Mahesh Arumugam received his B.E. degree in Computer Science and Engineering from College of Engineering, Guindy, Anna University in May 2001. In September 2003, he received his M.S. degree from Michigan State University, East Lansing, where he is currently a Ph.D. student. He is a member of the Software Engineering and Network Systems (SENS) Laboratory. His research interests include middleware services, program transformations, self-stabilization, and sensor networks. He is a student member of the IEEE and a member of the IEEE Computer Society. Contact him at arumugam@cse.msu.edu.

Reprinted from Trans. of the SICE
Vol. E-S-1 No. 1 33/42 2006